On affine usages in signal-based communication

Roberto M. Amadio, Mehdi Dogguy

On affine usages in signal-based communication

Roberto M. Amadio Mehdi Dogguy
Université Paris Diderot (Paris 7), PPS, UMR-7126

3rd September 2008

Abstract

We describe a type system for a *synchronous* π-calculus formalising the notion of *affine* usage in *signal-based* communication. In particular, we identify a limited number of usages that preserve affinity and that can be composed. As a main application of the resulting system, we show that typable programs are *deterministic*.

1 Introduction

We are interested in *synchronous* systems. In these systems, there is a notion of *instant* (or phase, or pulse, or round) and at each instant each component of the system, *a thread*, performs some actions and synchronizes with all the other threads. One may say that all threads proceed at the same speed and it is in this specific sense that we shall refer to *synchrony* in this work. *Signal-based* communication is often used as the basic interaction mechanism in synchronous systems (see, *e.g.*, [5, 6]). Signals play a role similar to *channels* in asynchronous systems. Our goal in this paper is to study the notion of *affine usage* in this context. In particular, we shall formalise our ideas in the context of a *synchronous* π-calculus ($S\pi$-calculus) introduced in [2]. We assume that the reader is familiar with the π-calculus and proceed to give a flavour of the language (the formal definition of the $S\pi$-calculus is recalled in section 2).

The syntax of the $S\pi$-calculus is similar to the one of the π-calculus, however there are some important *semantic* differences that we highlight in the following simple example. Assume $v_1 \neq v_2$ are two distinct values and consider the following program in $S\pi$:

$$P = \nu\ s_1, s_2\ (\ \overline{s_1}v_1 \mid \overline{s_1}v_2 \mid s_1(x).\ (s_1(y).\ (s_2(z).\ A(x,y)\ \underline{,B(!s_1))},\underline{0})\ \underline{,0}\ )$$

If we forget about the underlined parts and we regard s_1, s_2 as *channel names* then P could also be viewed as a π-calculus process. In this case, P would reduce to $P_1 = \nu s_1, s_2\ (s_2(z).A(\theta(x),\theta(y))$ where θ is a substitution such that $\theta(x), \theta(y) \in \{v_1, v_2\}$ and $\theta(x) \neq \theta(y)$. In $S\pi$, *signals persist within the instant* and P reduces to $P_2 = \nu s_1, s_2\ (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid (s_2(z).A(\theta(x),\theta(y)),\underline{B(!s_1)}))$ where again $\theta(x), \theta(y) \in \{v_1, v_2\}$ but possibly $\theta(x) = \theta(y)$. What happens next? In the π-calculus, P_1 is *deadlocked* and no further computation is possible. In the $S\pi$-calculus, the fact that no further computation is possible in P_2 is detected and marks the *end of the current instant*. Then an additional computation represented by the relation $\xrightarrow{N}$ moves P_2 to the following instant: $P_2 \xrightarrow{N} P_2' = \nu s_1, s_2\ B(v)$ where $v \in \{[v_1; v_2], [v_2; v_1]\}$. Thus at the end of the instant, a dereferenced signal such as $!s_1$ becomes a *list* (possibly empty) of (distinct) values emitted on s_1 during the instant and then all signals are reset.

We continue our informal discussion with an example of a 'server' handling a list of requests emitted in the previous instant on the signal s. For each request of the shape $\mathsf{req}(s', x)$, it provides an answer which is a function of x along the signal s' (the notation $x \trianglerighteq p$ is used to match a value x against a pattern p). The 'client' issues a request x on signal s and returns the reply on signal t.

$$\begin{array}{lcl} \mathit{Server}(s) & = & \mathsf{pause}.\mathit{Handle}(s, !s) \\ \mathit{Handle}(s, \ell) & = & [\ell \trianglerighteq \mathsf{cons}(\mathsf{req}(s', x), \ell')](\overline{s'}f(x) \mid \mathit{Handle}(s, \ell')), \mathit{Server}(s) \\ \mathit{Client}(x, s, t) & = & \nu s' \ (\overline{s}\mathsf{req}(s', x) \mid \mathsf{pause}.s'(x).\overline{t}x, 0) \ . \end{array}$$

Let us first notice that a request contains a 'pointer', namely the name of the signal on which to answer the request. Then the 'folklore solution' of transforming a list of values into one value via an associative and commutative function does not work here. Indeed there seems to be no reasonable way to define an associative and commutative function on pointers. Instead, we look at *Handle* as a function from (a signal and) a list of requests to behaviours which is invariant under permutations of the list of requests. Note that to express this invariance we need a notion of behavioural equivalence and that this equivalence must satisfy the usual associativity and commutativity laws of parallel composition and must be preserved by parallel composition.

These considerations are enough to argue that the *Server* is a 'deterministic' program. No matter how many clients will issue requests at each instant, the *Server* will provide an answer to each of them in the following instant in a way which is independent of the order of the requests. Let us now look at the *Client*. After issuing a request, the *Client* waits for a reply in the following instant. Clearly, if more than one reply comes, the outcome of the computation is not deterministic. For instance, we could have several 'Servers' running in parallel or a server could somehow duplicate the request. This means that the usage of the signal s must be such that many 'clients' may issue a request but at most one 'server' may handle them at the end of the instant in an 'affine' way. Further, on the client side, the return signal s' can only be used to read while on the server side it can only be used to emit.

This preliminary discussion suggests the need for a formal analysis of the principles that allow to establish the determinacy of a synchronous program. This analysis will be obviously inspired by previous work on the foundations of linear logic [7], on linear typing of functional programs (*e.g.*, [14]), and on linear usages of channels (*e.g.*, [10]). Following this line of works, the analysis presented in section 3 will take the form of a *typing system*. The previous section 2, will recall the formal definition of the $S\pi$-calculus. In the final section 4, first we shall introduce the properties of the typing system leading to a *subject reduction* theorem, and second we shall describe a suitable notion of typed bisimulation and show that with respect to this notion, typable programs can be regarded as *deterministic*.

2 Definition of the $S\pi$-calculus

We recall the formal definition of the $S\pi$-calculus and its bisimulation based semantics while referring the reader to [2, 4] for a deeper analysis. This section is rather technical but to understand the type system described in the following section 3 there are really just two points that the reader should keep in mind:

1. The semantics of the calculus is given by the labelled transition system presented in table 2. A reader familiar with a π-calculus with asynchronous communication can

understand these rules rather quickly. The main differences are (a) the rule for emitting a signal formalises the fact that a signal, unlike a channel, persists within an instant and (b) the rules that describe the computation at the end of the instant.

2. The labelled transition system induces a rather standard notion of bisimulation equivalence (definition 1) which is preserved by *static* contexts (fact 2).[1] In section 4, we shall introduce a 'typed' definition of the bisimulation and show that with respect to this definition, typable programs are deterministic.

2.1 Programs

Programs $P, Q, \ldots$ in the $S\pi$-calculus are defined in table 1. We use the notation $\mathbf{m}$ for a vector $m_1, \ldots, m_n$, $n \geq 0$. The informal behaviour of programs follows. 0 is the terminated thread. $A(\mathbf{e})$ is a (tail) recursive call of a thread identifier A with a vector $\mathbf{e}$ of expressions as argument; as usual the thread identifier A is defined by a unique equation $A(\mathbf{x}) = P$ such that the free variables of P occur in $\mathbf{x}$. $\overline{s}e$ evaluates the expression e and emits its value on the signal s. $s(x).P, K$ is the *present* statement which is the fundamental operator of the model [1]. If the values $v_1, \ldots, v_n$ have been emitted on the signal s then $s(x).P, K$ evolves non-deterministically into $[v_i/x]P$ for some v_i ($[\_/\_]$ is our notation for substitution). On the other hand, if no value is emitted then the continuation K is evaluated at the end of the instant. $[s_1 = s_2]P_1, P_2$ is the usual matching function of the π-calculus that runs P_1 if s_1 equals s_2 and P_2, otherwise. Here both s_1 and s_2 are free. $[u \trianglerighteq p]P_1, P_2$, matches u against the pattern p. We assume u is either a variable x or a value v and p has the shape $\mathsf{c}(\mathbf{x})$, where c is a constructor and $\mathbf{x}$ is a vector of distinct variables. We also assume that if u is a variable x then x does not occur free in P_1. At run time, u is always a *value* and we run θP_1 if $\theta = match(u, p)$ is the substitution matching u against p, and P_2 if the substitution does not exist (written $match(u, p) \uparrow$). Note that as usual the variables occurring in the pattern p (including signal names) are bound in P_1. $\nu s\ P$ creates a new signal name s and runs P. $(P_1 \mid P_2)$ runs in parallel P_1 and P_2. A continuation K is simply a recursive call whose arguments are either expressions or values associated with signals at the end of the instant in a sense that we explain below. We shall also write $\mathsf{pause}.K$ for $\nu s\ s(x).0, K$ with s not free in K. This is the program that waits till the end of the instant and then evaluates K.

2.2 Expressions

Expressions are partitioned in several syntactic categories as specified in table 1. As in the π-calculus, signal names stand both for signal constants as generated by the ν operator and signal variables as in the formal parameter of the present operator. Variables *Var* include signal names as well as variables of other types. Constructors *Cnst* include $*$, nil, and cons. Values *Val* are terms built out of constructors and signal names. Patterns *Pat* are terms built out of constructors and variables (including signal names). If P, p are a program and a pattern then we denote with $fn(P), fn(p)$ the set of free signal names occurring in them, respectively. We also use $FV(P), FV(p)$ to denote the set of free variables (including signal names). We assume first-order function symbols $f, g, \ldots$ and an evaluation relation $\Downarrow$ such that for every function symbol f and values $v_1, \ldots, v_n$ of suitable type there is a unique value

[1] As a matter of fact the labelled transition system is built so that the definition of bisimulation equivalence looks standard [4].

P	$::= 0 \mid A(\mathbf{e}) \mid \overline{s}e \mid s(x).P, K \mid$	(programs)
	$\quad [s_1 = s_2]P_1, P_2 \mid [u \trianglerighteq p]P_1, P_2 \mid \nu s\ P \mid P_1 \mid P_2$	
K	$::= A(\mathbf{r})$	(continuation next instant)
Sig	$::= s \mid t \mid \cdots$	(signal names)
Var	$::= Sig \mid x \mid y \mid z \mid \cdots$	(variables)
$Cnst$	$::= * \mid \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{c} \mid \mathsf{d} \mid \cdots$	(constructors)
Val	$::= Sig \mid Cnst(Val, \ldots, Val)$	(values $v, v', \ldots$)
Pat	$::= Cnst(Var, \ldots, Var)$	(patterns $p, p', \ldots$)
Fun	$::= f \mid g \mid \cdots$	(first-order function symbols)
Exp	$::= Var \mid Cnst(Exp, \ldots, Exp) \mid Fun(Exp, \ldots, Exp)$	(expressions $e, e', \ldots$)
$Rexp$	$::= !Sig \mid Var \mid Cnst(Rexp, \ldots, Rexp) \mid$	
	$\quad Fun(Rexp, \ldots, Rexp)$	(exp. with deref. $r, r', \ldots$)

Table 1: Syntax of programs and expressions

v such that $f(v_1, \ldots, v_n) \Downarrow v$ and $fn(v) \subseteq \bigcup_{i=1,\ldots,n} fn(v_i)$. Expressions *Exp* are terms built out of variables, constructors, and function symbols. The evaluation relation $\Downarrow$ is extended in a standard way to expressions whose only free variables are signal names. Finally, *Rexp* are expressions that may include the value associated with a signal s at the end of the instant (which is written $!s$, following the ML notation for dereferenciation). Intuitively, this value is a *list of values* representing the set of values emitted on the signal during the instant.

The definition of a *simple* type system for the $S\pi$-calculus can be extracted from the more elaborate type system presented in section 3 by confusing 'set-types' with 'list-types' and by neglecting all considerations on usages.

2.3 Actions

The syntactic category *act* of *actions* described in table 2 comprises relevant, auxiliary, and nested actions. The operations *fn* (free names), *bn* (bound names), and *n* (both free and bound names) are defined as in the π-calculus [13].

The *relevant actions* are those that are actually considered in the bisimulation game. They consist of: (i) an internal action τ, (ii) an emission action $\nu\mathbf{t}\ \overline{s}v$ where it is assumed that the signal names $\mathbf{t}$ are distinct, occur in v, and differ from s, (iii) an input action sv, and (iv) an action N (for *Next*) that marks the move from the current to the next instant.

The *auxiliary actions* consist of an input action $s?v$ which is coupled with an emission action in order to compute a τ action and an action (E, V) which is just needed to compute an action N. The latter is an action that can occur *exactly* when the program cannot perform τ actions and it amounts to (i) collect in lists the set of values emitted on every signal, (ii) to reset all signals, and (iii) to initialise the continuation K for each present statement of the shape $s(x).P, K$.

In order to formalise these three steps we need to introduce some notation. Let E vary over functions from signal names to finite sets of values. Denote with $\emptyset$ the function that associates the empty set with every signal name, with $[M/s]$ the function that associates the set M with the signal name s and the empty set with all the other signal names, and with $\cup$ the union of functions defined point-wise.

We represent a set of values as a list of the values belonging to the set. More precisely, we write $v \,\|\!-\! M$ and say that v *represents* M if $M = \{v_1, \ldots, v_n\}$ and $v = [v_{\pi(1)}; \ldots; v_{\pi(n)}]$ for some permutation π over $\{1, \ldots, n\}$. Suppose V is a function from signal names to lists

of values. We write $V \Vdash E$ if $V(s) \Vdash E(s)$ for every signal name s. We also write $dom(V)$ for $\{s \mid V(s) \neq []\}$. If K is a continuation, *i.e.*, a recursive call $A(\mathbf{r})$, then $V(K)$ is obtained from K by replacing each occurrence $!s$ of a dereferenced signal with the associated value $V(s)$. We denote with $V[\ell/s]$ the function that behaves as V except on s where $V[\ell/s](s) = \ell$.

With these conventions, a transition $P \xrightarrow{(E,V)} P'$ intuitively means that (1) P is suspended, (2) P emits exactly the values specified by E, and (3) the behaviour of P in the following instant is P' and depends on V. It is convenient to compute these transitions on programs where all name generations are lifted at top level. We write $P \succeq Q$ if we can obtain Q from P by repeatedly transforming, for instance, a subprogram $\nu s P' \mid P''$ into $\nu s(P' \mid P'')$ where $s \notin fn(P'')$.

Finally, the *nested actions* $\mu, \mu', \ldots$ are certain actions (either relevant or auxiliary) that can be produced by a sub-program and that we need to propagate to the top level.

2.4 Labelled transition system and bisimulation

The labelled transition system is defined in table 2 where rules apply to programs whose only free variables are signal names and with standard conventions on the renaming of bound names. As usual, one can rename bound variables, and symmetric rules are omitted. The first 12 rules from (*out*) to (ν_{ex}) are quite close to those of a polyadic π-calculus with asynchronous communication (see [8, 3]) with the following exception: rule (*out*) models the fact that the emission of a value on a signal *persists* within the instant. The last 5 rules from (0) to (*next*) are quite specific of the $S\pi$-calculus and determine how the computation is carried on at the end of the instant (cf. discussion in 2.3).

We derive from the labelled transition system a notion of (weak) labelled bisimulation. First define $\stackrel{\alpha}{\Rightarrow}$ as $(\stackrel{\tau}{\rightarrow})^*$ if $\alpha = \tau$, $(\stackrel{\tau}{\Rightarrow}) \circ (\stackrel{N}{\rightarrow})$ if $\alpha = N$, and $(\stackrel{\tau}{\Rightarrow}) \circ (\stackrel{\alpha}{\rightarrow}) \circ (\stackrel{\tau}{\Rightarrow})$ otherwise. This is the standard definition except that we insist on *not* having internal reductions after an N action. Intuitively, we assume that an observer can control the execution of programs so as to be able to test them at the very beginning of each instant. We write $P \stackrel{\alpha}{\rightarrow} \cdot$ for $\exists P' \ (P \stackrel{\alpha}{\rightarrow} P')$.

Definition 1 (labelled bisimulation) *A symmetric relation $\mathcal{R}$ on programs is a labelled bisimulation if $P \mathcal{R} Q$, $P \stackrel{\alpha}{\rightarrow} P'$, $bn(\alpha) \cap fn(Q) = \emptyset$ implies $\exists Q' \ (\ Q \stackrel{\alpha}{\Rightarrow} Q', \quad P' \mathcal{R} Q'\ )$. We denote with $\approx$ the largest labelled bisimulation.*

Fact 2 ([4]) *Labelled bisimulation is preserved by parallel composition and name generation.*

3 An affine type system

An analysis of the notion of determinacy carried on in [4], along the lines of [12], suggests that there are basically two situations that need to be analysed in order to guarantee the determinacy of programs. (1) At least two distinct values compete to be received within an instant, for instance, consider: $\overline{s}v_1 \mid \overline{s}v_2 \mid s(x).P, K$. (2) At the end of the instant, at least two distinct values are available on a signal. For instance, consider: $\overline{s}v_1 \mid \overline{s}v_2 \mid \mathsf{pause}.A(!s)$. A sensible approach is to avoid completely the first situation and to allow the second provided the behaviour of the continuation A does not depend on the order in which the values are collected. Technically, we consider a notion of *affine signal usage* to guarantee the first condition and a notion of *set type* for the second one. While this is a good starting point,

$$
\begin{array}{lll}
act & ::= \alpha \mid aux & \text{(actions)} \\
\alpha & ::= \tau \mid \nu\mathbf{t}\ \overline{s}v \mid sv \mid N & \text{(relevant actions)} \\
aux & ::= s?v \mid (E, V) & \text{(auxiliary actions)} \\
\mu & ::= \tau \mid \nu\mathbf{t}\ \overline{s}v \mid s?v & \text{(nested actions)}
\end{array}
$$

$$(out)\quad \frac{e \Downarrow v}{\overline{s}e \xrightarrow{\overline{s}v} \overline{s}e} \qquad (in_{aux})\quad \frac{}{s(x).P, K \xrightarrow{s?v} [v/x]P}$$

$$(in)\quad \frac{}{P \xrightarrow{sv} (P \mid \overline{s}v)} \qquad (rec)\quad \frac{A(\mathbf{x}) = P, \quad \mathbf{e} \Downarrow \mathbf{v}}{A(\mathbf{e}) \xrightarrow{\tau} [\mathbf{v}/\mathbf{x}]P}$$

$$(=_1^{sig})\quad \frac{}{[s = s]P_1, P_2 \xrightarrow{\tau} P_1} \qquad (=_2^{sig})\quad \frac{s_1 \neq s_2}{[s_1 = s_2]P_1, P_2 \xrightarrow{\tau} P_2}$$

$$(=_1^{ind})\quad \frac{match(v, p) = \theta}{[v \trianglerighteq p]P_1, P_2 \xrightarrow{\tau} \theta P_1} \qquad (=_1^{ind})\quad \frac{match(v, p) = \uparrow}{[v \trianglerighteq p]P_1, P_2 \xrightarrow{\tau} P_2}$$

$$(comp)\quad \frac{P_1 \xrightarrow{\mu} P_1' \quad bn(\mu) \cap fn(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2} \qquad (synch)\quad \frac{P_1 \xrightarrow{\nu\mathbf{t}\ \overline{s}v} P_1' \quad P_2 \xrightarrow{s?v} P_2' \quad \{\mathbf{t}\} \cap fn(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau} \nu\mathbf{t}\ (P_1' \mid P_2')}$$

$$(\nu)\quad \frac{P \xrightarrow{\mu} P' \quad t \notin n(\mu)}{\nu t\ P \xrightarrow{\mu} \nu t\ P'} \qquad (\nu_{ex})\quad \frac{P \xrightarrow{\nu\mathbf{t}\ \overline{s}v} P' \quad t' \neq s \quad t' \in n(v) \backslash \{\mathbf{t}\}}{\nu t'\ P \xrightarrow{(\nu t', \mathbf{t})\overline{s}v} P'}$$

$$(0)\quad \frac{}{0 \xrightarrow{\emptyset, V} 0} \qquad (reset)\quad \frac{e \Downarrow v \quad v \text{ occurs in } V(s)}{\overline{s}e \xrightarrow{[\{v\}/s], V} 0}$$

$$(cont)\quad \frac{s \notin dom(V)}{s(x).P, K \xrightarrow{\emptyset, V} V(K)} \qquad (par)\quad \frac{P_i \xrightarrow{E_i, V} P_i' \quad i = 1, 2}{(P_1 \mid P_2) \xrightarrow{E_1 \cup E_2, V} (P_1' \mid P_2')}$$

$$(next)\quad \frac{P \succeq \nu\mathbf{s}\ P' \quad V \Vdash E \quad P' \xrightarrow{E, V} P''}{P \xrightarrow{N} \nu\mathbf{s}\ P''}$$

Table 2: Labelled transition system

it falls short of providing a completely satisfying answer because the type constructions do *not* compose very well. Then our goal is to discover a collection of *signal usages* with better compositionality properties. The outcome of our analysis are three new kinds of usages (kinds $3-5$ in table 3).

3.1 Usages

In first approximation, we may regard a *usage* as an element of the set $L = \{0, 1, \infty\}$ with the intuition that 0 corresponds to no usage at all, 1 to at most one usage, and ∞ to any usage. We *add* usages with a *partial* operation $\oplus$ such that $0 \oplus a = a \oplus 0 = a$ and $\infty \oplus \infty = \infty$, and which is undefined otherwise (note in particular that $1 \oplus 1$ is undefined). The addition induces an *order* by $a \leq b$ if $\exists\, c \;\; a \oplus c = b$. With respect to this order, 0 is the least element while 1 and ∞ are *incomparable*. If $a \geq b$ then we define a *subtraction* operation $a \ominus b$ as the *largest* c such that $a = b \oplus c$. Therefore: $a \ominus 0 = a$, $1 \ominus 1 = 0$, and $\infty \ominus \infty = \infty$.

This classification of usages is adequate when handling purely functional data where the intuition is that data with usage 1 have at most one pointer to them [14]. However, when handling more complex entities such as references, channels, or signals it is convenient to take a more refined view. Specifically, a usage can be refined to include information about whether a signal is used: (i) to emit, (ii) to receive during the instant, or (iii) to receive at the end of the instant. Then a usage becomes an element of L^3. Among the 27 possible usages of the shape (a, b, c) for $a, b, c \in L$, we argue that there are 5 *main* ones as described in table 3 (left part). First of all, we must have $a \neq 0$ and $(b \neq 0 \vee c \neq 0)$ since a signal on which we cannot send or receive has no interest. Now if $a = \infty$ then we are forced to take $b = 0$ since we want to preserve the determinacy. Then for $c = \infty$ we have the usage e_1 and for $c = 1$ we have the usage e_3. Suppose now $a = 1$. One choice is to have $b = c = \infty$ and then we have the usage e_2. On the other hand if we want to preserve affinity then we should receive the emitted value at most once. Hence we have $b = 0, c = 1$ or $b = 1, c = 0$ which correspond to the usages e_4 and e_5, respectively. From these 5 *main* usages within an instant, we obtain the *derived ones* (see again table 3) by simply turning one or more 1's to 0's. We only add, subtract, compare usages in L^3 that are derived from the same main usage.

In a *synchronous* framework, it makes sense to consider how usages vary over *time*. The *simplest* solution would be to look at signal usages of the shape x^ω, $x \in L^3$, which are *invariant* under time. However, to reason effectively on programs, we are led to consider signal usages of the shape xy^ω where $x, y \in L^3$ are derived from the same main usage.

The reader may have noticed that in this discussion we have referred to increasingly complex 'usages' varying over L, L^3, and $(L^3)^\omega$. Henceforth a signal usage belongs to $(L^3)^\omega$. Usages are classified in 5 *kinds* as showed in table 3. [2]

We denote with U the set of all these usages and with $U(i)$ the set of usages of kind i, for $i = 1, \ldots, 5$. We consider that the addition operation $\oplus$ is defined only if $u, u' \in U(i)$ and $u \oplus u' \in U(i)$ for some $i \in \{1, \ldots, 5\}$. Similar conventions apply when comparing and subtracting usages. If $u \in U$ then $\uparrow u$, the *shift* of u, is the infinite word in U obtained from u by removing the first character. This operation is always defined. If u is a signal usage, then $u(i)$ for $i \geq 0$ denotes its i^{th} character and $u(i)_j$ for $j \in \{1, 2, 3\}$ the j^{th} component of $u(i)$.

We classify the usages according to 3 properties: affinity, uniformity, and preservation of affinity. We say that a usage is *affine* if it contains a '1′ and *non-affine* otherwise. We also

[2] The fact that, *e.g.*, $(1, 0, 0)$ occurs both in the usages of kind 4 and 5 is a slight source of ambiguity which is resolved by assuming that the kind of the usage is made explicit.

main usages	derived usages	$xy^\omega \in U(i)$ is	affine	uniform	aff. preserving
$e_1 = (\infty, 0, \infty)$	-	$i = 1$	*no*	*yes*	*no*
$e_2 = (1, \infty, \infty)$	$(0, \infty, \infty)$	$i = 2$	*yes/no*	*yes/no*	*no*
$e_3 = (\infty, 0, 1)$	$(\infty, 0, 0)$	$i = 3$	*yes/no*	*yes/no*	*yes*
$e_4 = (1, 0, 1)$	$(1, 0, 0), (0, 0, 1), (0, 0, 0)$	$i = 4$	*yes/no*	*yes/no*	*yes*
$e_5 = (1, 1, 0)$	$(1, 0, 0), (0, 1, 0), (0, 0, 0)$	$i = 5$	*yes/no*	*yes/no*	*yes*

Table 3: Usages and their classification

say that it is *uniform* if it is of the shape x^ω and that it is *neutral* if it is the neutral element with respect to the addition $\oplus$ on the set of usages $U(i)$ to which it belongs. It turns out that the non-affine signal usages are always uniform and moreover they coincide with the neutral ones. Finally, by definition, the usages in the sets $U(i)$ for $i = 3, 4, 5$ are *affine preserving* The classification is summarised in the table 3 (right part).

3.2 Types

In first approximation, types are either *inductive types* or *signal* types. As usual, an inductive type such as the type $List(\sigma)$ of lists of elements of type σ is defined by an equation $List(\sigma) = \mathsf{nil} \mid \mathsf{cons}\ of\ \sigma, List(\sigma)$ specifying the ways in which an element of this type can be built.

In our context, inductive types come with a *usage* x which belongs to the set $\{1, \infty\}$ and which intuitively specifies whether the values of this type can be used at most once or arbitrarily many times (once more we recall that 1 and ∞ are incomparable). To summarise, if $\sigma_1, \ldots, \sigma_k$ are types already defined then an inductive type $C_x(\sigma_1, \ldots, \sigma_k)$ is defined by case on constructors of the shape $\mathsf{c}\ of\ \sigma'_1, \ldots, \sigma'_m$ where the types σ'_j, $j = 1, \ldots, m$ are either one of the types σ_i, $i = 1, \ldots, n$ or the inductive type $C_x(\ldots)$ being defined. There is a further constraint that has to be respected, namely that if one of the types σ_i is 'affine' then the usage x must be affine preserving, *i.e.*, $x = 1$. An affine type is simply a type which contains an affine usage. The grammar in table 4 will provide a precise definition of the affine types.

When collecting the values at the end of the instant we shall also need to consider *set types*. They are described by an equation $Set_x(\sigma) = \mathsf{nil} \mid \mathsf{cons}\ of\ \sigma, Set_x(\sigma)$ which is quite similar to the one for lists. Note that set types too come with a usage $x \in \{1, \infty\}$ and that if σ is an affine type then the usage x must be affine preserving. The reader might have noticed that we take the freedom of using the constructor nil both with the types $List_u(\sigma)$ and $Set_u(\sigma)$, $u \in \{1, \infty\}$, and the constructor cons both with the types $(\sigma, List_u(\sigma)) \to List_u(\sigma)$ and $(\sigma, Set_u(\sigma)) \to Set_u(\sigma)$. However, one should assume that a suitable label on the constructors will allow to disambiguate the situation.

Finally, we denote with $Sig_u(\sigma)$ the type of signals carrying values of type σ according to the signal usage u. As for inductive and set types, if σ is an affine type then the signal usage u must be affine preserving. To formalise these distinctions, we are lead to use several names for types as specified in table 4. We denote with κ non-affine (or classical) types, *i.e.*, types that carry *no* affine information. These types have a uniform usage. We denote with λ affine and uniform types. The types $\sigma, \sigma', \ldots$ stand for types with uniform usage (either non-affine or affine). Finally, the types $\rho, \rho', \ldots$ include all the previous ones plus types that have a non-uniform usage. We notice that classical uniform types can be nested in an arbitrary way, while affine uniform types can only be nested under type constructors that preserve affinity.

Moreover, types with non-uniform usages (either classical or affine) cannot be nested at all.[3]

The partial operation of addition $\oplus$ is extended to types so that: $Op_{u_1}(\sigma) \oplus Op_{u_2}(\sigma) = Op_{u_1 \oplus u_2}(\sigma)$, where Op can be C, Set, or Sig, and provided that $u_1 \oplus u_2$ is defined. For instance, $List_1(\lambda) \oplus List_1(\lambda)$ is undefined because $1 \oplus 1$ is not defined.

A type context (or simply a context) Γ is a partial function with finite domain $dom(\Gamma)$ from variables to types. An addition operation $\Gamma_1 \oplus \Gamma_2$ on contexts is defined, written $(\Gamma_1 \oplus \Gamma_2)\downarrow$, if and only if for all x such that $\Gamma_1(x) = \rho_1$ and $\Gamma_2(x) = \rho_2$, the type $\rho_1 \oplus \rho_2$ is defined. The shift operation is extended to contexts so that $(\uparrow \Gamma)(x) = Sig_{(\uparrow u)}(\sigma)$ if $\Gamma(x) = Sig_u(\sigma)$ and $(\uparrow \Gamma)(x) = \Gamma(x)$ otherwise. We also denote with $\Gamma, x : \sigma$ the context Γ *extended* with the pair $x : \sigma$ (so $x \notin dom(\Gamma)$). We say that a context is *neutral* (*uniform*) if it assigns to variables neutral (uniform) types.

3.3 Semantic instrumentation

As we have seen, each signal belongs to exactly one of 5 kinds of usages. Let us consider in particular the kind 5 whose main usage is e_5. The forthcoming type system is supposed to guarantee that a value emitted on a signal of kind 5 is received at most once during an instant. Now, consider the program $\overline{s}t \mid s(x).\overline{x}, 0$ and attribute a usage e_5^ω to the signals s and t. According to this usage this program should be well typed. However, if we apply the labelled transition system in table 2, this program reduces to $(\overline{s}t \mid \overline{t})$ which fails to be well-typed because the double occurrence of t is not compatible with an affine usage of t. Intuitively, after the signal s has been read once no other synchronisation should arise during the instant either within the program or with the environment. To express this fact we proceed as follows. First, we instrument the semantics so that it marks (underlines) the emissions on signals of kind 5 that have been used at least once during the instant. The emission has no effect on the labelled transition system in the sense that $\underline{\overline{s}e}$ behaves exactly as $\overline{s}e$.

$$(out)\quad \frac{e \Downarrow v}{\overline{s}e \xrightarrow{\overline{s}v} \underline{\overline{s}e}} \qquad (\underline{out})\quad \frac{e \Downarrow v}{\underline{\overline{s}e} \xrightarrow{\overline{s}v} \underline{\overline{s}e}} \qquad (\underline{reset})\quad \frac{e \Downarrow v \quad v \text{ occurs in } V(s)}{\underline{\overline{s}e} \xrightarrow{[\{v\}/s],V} 0}$$

On the other hand, we introduce a special rule ($\underline{out}$) to type $\underline{\overline{s}e}$ which requires at least a usage $(1,1,0) \cdot (0,0,0)^\omega$ for the signal s while neglecting the expression e. By doing this, we make sure that a second attempt to receive on s will produce a type error. In other terms, if typing is preserved by 'compatible' transitions, then we can be sure that a value emitted on a signal of kind 5 is received at most once within an instant.

3.4 Type system

The type system is built around few basic ideas. (1) Usages including both input and output capabilities can be decomposed in simpler ones. For instance, $(1,1,0)^\omega = (1,0,0)(0,1,0)^\omega \oplus (0,1,0)(1,0,0)^\omega$. (2) A rely-guarantee kind of reasoning: when we emit a value we *guarantee* certain resources while when we receive a value we *rely* on certain resources. (3) Every affine usage can be consumed at most once in the typing judgement (and in the computation).

[3] What's the meaning of sending a data structure containing informations whose usage is time-dependent? Is the time information relative to the instant where the data structure is sent or used? We leave open the problem of developing a type theory with usages more complex than the ones of the shape xy^ω considered here.

When formalising the typing judgements we need to distinguish the typing of an expression e from the typing of an expression with dereferenciation r and the typing of a recursive call $A(e_1,\ldots,e_n)$ from the typing of a recursive call at the end of the instant $A(r_1,\ldots,r_n)$. To do this we shall write $[r]$ rather than r and $[A(r_1,\ldots,r_n)]$ rather than $A(r_1,\ldots,r_n)$.

We shall consider *four typing judgements*: $\Gamma \vdash e : \rho$, $\Gamma \vdash [r] : \rho$, $\Gamma \vdash P$, and $\Gamma \vdash [A(r_1,\ldots,r_n)]$, and we wish to refer to them with a *uniform* notation $\Gamma \vdash U : T$. To this end, we introduce a fictious type Pr of programs and regard the judgements $\Gamma \vdash P : Pr$ and $\Gamma \vdash [A(r_1,\ldots,r_n)] : Pr$ as an expansion of $\Gamma \vdash P$ and $\Gamma \vdash [A(r_1,\ldots,r_n)]$, respectively. Then we let U stand for one of e, $[r]$, P, $[A(r_1,\ldots,r_n)]$, and T for one of ρ, Pr.

We assume that function symbols are given non-affine types of the shape $(\kappa_1,\ldots,\kappa_n) \to \kappa$. We denote with k either a constructor or a function symbol and we assume that its type is explicitly given.

The typing rules are given in table 4. We comment first on the typing rules for the expressions. We notice that the arguments and the result of a constructor or a function symbol have always a uniform type. The rules $(!_{Set})$ and $(!_{List})$ describe the type of a dereferenced signal following its usage. If the usage is of kind 1 then the list of values associated with the signal at the end of the instant must be treated as a set, if the usage is of kind 2 then we know that the list of values contains at most one element and therefore its processing will certainly be 'order-independent', if the usage is of kind 3 then the list may contain several values and it must be processed as an *affine* set, finally if the usage is of kind 4 (the usage of kind 5 forbids reception at the end of the instant) then again the list of values will contain at most one element so we can rely on an *affine* list type.

Notice the special form of the rule $[var_{sig}]$. The point here is that in a recursive call $K = A(!s, s)$ at the end of instant, we need to distinguish the resources needed to type $!s$ which should relate to the *current* instant from the resources needed to type s which should relate to the *following instants*. For instance, we want to type K in a context $s : Sig_u(\sigma)$ where $u = (0,0,1)^\omega$. This is possible because we can decompose u in $u_1 \oplus u_2$, where $u_1 = (0,0,1)(0,0,0)^\omega$ and $u_2 = (0,0,0)(0,0,1)^\omega$, and we can rely on u_1 to type $[!s]$ and on u_2 to type $[s]$ (by $[var_{sig}]$).

A set-type is a particular case of quotient type and therefore its definition goes through the definition of an equivalence relation $\sim_\rho$ on values. This is defined as the least equivalence relation such that $s \sim_{Sig_u(\sigma)} s$, $\mathsf{c} \sim_{C(\sigma)} \mathsf{c}$, if c is a constant of type $C(\sigma)$, and

$$\begin{array}{ll}
\mathsf{c}(v_1,\ldots,v_n) \sim_{C_u(\sigma_1,\ldots,\sigma_n)} \mathsf{c}(u_1,\ldots,u_n) & \text{if } v_i \sim_{\sigma_i} u_i \text{ for } i = 1,\ldots,n \\
[v_1;\ldots;v_n] \sim_{Set_u(\sigma)} [u_1;\ldots;u_m] & \text{if } \{v_1,\ldots,v_n\} \sim_{Set_u(\sigma)} \{u_1,\ldots,u_m\}, \\
\text{where: } \{v_1,\ldots,v_n\} \sim_{Set_u(\sigma)} \{u_1,\ldots,u_m\} & \text{if for a permutation } \pi, v_i \sim_\sigma u_{\pi(i)} \ .
\end{array}$$

Furthermore, we assume that each function symbol f, coming with a (classical) type $(\kappa_1,\ldots,\kappa_n) \to \kappa$, *respects* the typing in the following sense: (1) if $v_i \sim_{\kappa_i} u_i$, $i = 1,\ldots,n$, $f(v_1,\ldots,v_n) \Downarrow v$ and $f(u_1,\ldots,u_n) \Downarrow u$ then $v \sim_\kappa u$. (2) If $\Gamma \vdash f(v_1,\ldots,v_n) : \kappa$ and $f(v_1,\ldots,v_n) \Downarrow v$ then $\Gamma \vdash v : \kappa$.

Finally, we turn to the typing of programs. We assume that each thread identifier A, defined by an equation $A(x_1,\ldots,x_n) = P$, comes with a type $(\sigma_1,\ldots,\sigma_n)$. Hence we require these types to be uniform. We also require that A has the property that: (i) if $v_i \sim_{\sigma_i} u_i$ for $i = 1,\ldots,n$ then $A(v_1,\ldots,v_n) \approx A(u_1,\ldots,u_n)$ and (ii) $x_1 : \sigma_1,\ldots,x_n : \sigma_n \vdash P$ is derivable.

We also suppose that generated signals names are *explicitly* labelled with their types as in $\nu s : \rho\ P$. The labelled transition system in table 2 is adapted so that the output action carries the information on the types of the extruded names. This type is lifted by the rule

$$
\begin{array}{lll}
\kappa & ::= C_\infty(\kappa) \mid Set_\infty(\kappa) \mid Sig_u(\kappa) & (u \text{ neutral}) \\
\lambda & ::= C_1(\sigma) \mid Set_1(\sigma) \mid Sig_u(\kappa) \mid Sig_v(\lambda) & (u \text{ affine and uniform}, v \text{ aff.-pres. and uniform}) \\
\sigma & ::= \kappa \mid \lambda & (\text{uniform types}) \\
\rho & ::= \sigma \mid Sig_u(\kappa) \mid Sig_v(\lambda) & (v \text{ affine-preserving})
\end{array}
$$

$$(var)\quad \frac{u \geq u' \quad Op \in \{Sig, Set, C\}}{\Gamma, x : Op_u(\sigma) \vdash x : Op_{u'}(\sigma)} \qquad (k)\quad \frac{\Gamma_i \vdash e_i : \sigma_i \quad i = 1, \ldots, n \quad k : (\sigma_1, \ldots, \sigma_n) \to \sigma \quad k = f \text{ or } k = \mathsf{c}}{\Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash k(e_1, \ldots, e_n) : \sigma}$$

$$[var_C]\quad \frac{Op = C \quad Op = Set}{\Gamma, x : Op_u(\sigma) \vdash [x] : Op_u(\sigma)} \qquad [var_{sig}]\quad \frac{y^\omega \geq u}{\Gamma, s : Sig_{xy^\omega}(\sigma) \vdash [s] : Sig_u(\sigma)}$$

$$[k]\quad \frac{\Gamma_i \vdash [r_i] : \sigma_i \quad i = 1, \ldots, n \quad k : (\sigma_1, \ldots, \sigma_n) \to \sigma \quad k = f \text{ or } k = \mathsf{c}}{\Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash [k(r_1, \ldots, r_n)] : \sigma}$$

$$[!_{Set}]\quad \frac{(u(0) \geq (\infty, 0, \infty) \ \wedge \ x = \infty) \vee (u(0) \geq (\infty, 0, 1) \ \wedge \ x = 1)}{\Gamma, s : Sig_u(\sigma) \vdash [!s] : Set_x(\sigma)} \qquad [!_{List}]\quad \frac{(u(0) \geq (0, \infty, \infty) \ \wedge \ x = \infty) \vee (u(0) \geq (0, 0, 1) \ \wedge \ x = 1)}{\Gamma, s : Sig_u(\sigma) \vdash [!s] : List_x(\sigma)}$$

$$(0)\quad \frac{}{\Gamma \vdash 0} \qquad (out)\quad \frac{\Gamma_1 \vdash s : Sig_u(\sigma) \quad u(0)_1 \neq 0 \quad \Gamma_2 \vdash e : \sigma}{\Gamma_1 \oplus \Gamma_2 \vdash \overline{s}e}$$

$$(\nu)\quad \frac{\Gamma, s : Sig_u(\sigma) \vdash P}{\Gamma \vdash \nu s : Sig_u(\sigma)\ P} \qquad (in)\quad \frac{\Gamma_1 \vdash s : Sig_u(\sigma) \quad u(0)_2 \neq 0 \quad \Gamma_2, x : \sigma \vdash P \quad (\Gamma_1 \oplus \Gamma_2) \vdash [A(\mathbf{r})]}{(\Gamma_1 \oplus \Gamma_2) \vdash s(x).P, A(\mathbf{r})}$$

$$(m_s)\quad \frac{s_1, s_2 \in dom(\Gamma) \quad \Gamma \vdash P_i \quad i = 1, 2}{\Gamma \vdash [s_1 = s_2]P_1, P_2} \qquad (m_\mathsf{c})\quad \frac{\mathsf{c} : (\sigma_1, \ldots, \sigma_n) \to \sigma \quad \Gamma_1 \vdash u : \sigma \quad \Gamma_2, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1 \quad (\Gamma_1 \oplus \Gamma_2) \vdash P_2}{\Gamma_1 \oplus \Gamma_2 \vdash [u \trianglerighteq \mathsf{c}(x_1, \ldots, x_n)]P_1, P_2}$$

$$(par)\quad \frac{\Gamma_i \vdash P_i \quad i = 1, 2}{\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2} \qquad (rec)\quad \frac{A : (\sigma_1, \ldots, \sigma_n), \quad \Gamma_i \vdash e_i : \sigma_i \quad i = 1, \ldots, n}{\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash A(e_1, \ldots, e_n)}$$

$$(\underline{out})\quad \frac{\Gamma \vdash s : Sig_u(\sigma) \quad u(0) = (1, 1, 0)}{\Gamma \vdash \underline{\overline{s}e}} \qquad [rec]\quad \frac{A : (\sigma_1, \ldots, \sigma_n), \quad \Gamma_i \vdash [r_i] : \sigma_i \quad i = 1, \ldots, n}{\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash [A(r_1, \ldots, r_n)]}$$

Table 4: Affine type system

(*next*) so that, *e.g.*, $\nu s : \rho\ s.0, A(s) \xrightarrow{N} \nu s :\uparrow \rho\ A(s)$.

Example 3 *With reference to the example of client-server in section 1, assume an inductive (non-affine) type D of data. Let $\sigma_1 = Sig_{u_1}(D)$ where $u_1 = (1,0,0)^\omega$ be the type of the signals on which the server will eventually provide an answer. Let $Req_1(\sigma_1, D) =$ req σ_r, D be the type of requests which are pairs composed of a signal and a datum. Let $\sigma_{set} = Set_1(Req_1(\sigma_1, D))$ be the type of the set of requests issued by the clients. Let $\sigma = Sig_u(Req_1(\sigma_1, D))$ with $u = (\infty, 0, 1)^\omega$ be the type of the signal on which the server gets the requests and $\sigma' = Sig_{u'}(Req_1(\sigma_1, D))$, with $u' = (\infty, 0, 0)^\omega$, the related type of the signal on which the clients send the requests. Finally, let $\sigma_t = Sig_u(D)$ be the type of the signal on which the client sends the received answer (with a suitable usage u). Then we can type Server and Client as follows: Server $: (\sigma)$, Handle $: (\sigma, \sigma_{set})$, and Client $: (D, \sigma', \sigma_t)$.*

Remark 4 *In a practical implementation of the type system, one can expect the programmer to assign a kind $(1-5)$ to each signal and let the system infer a* minimum *usage which is compatible with the operations performed by the program.*

4 Results

We start by stating the expected *weakening* and *substitution* properties of the type system.

Lemma 5 (weakening) *If $\Gamma \vdash U : T$ and $(\Gamma \oplus \Gamma') \downarrow$ then $(\Gamma \oplus \Gamma') \vdash U : T$.*

Lemma 6 (substitution) *If $\Gamma, x : \rho \vdash U : T$, $\Gamma' \vdash v : \rho$, and $(\Gamma \oplus \Gamma') \downarrow$ then $(\Gamma \oplus \Gamma') \vdash [v/x]U : T$.*

Next we specify when a context Γ is *compatible* with an action *act*, written $(\Gamma, act) \downarrow$. Recall that V and E denote a function from signals to finite lists of distinct values and finite sets of values, respectively. If $V(s) = [v_1; \ldots; v_n]$ then let $(V \backslash E)(s) = \{v_1, \ldots, v_n\} \backslash E(s)$. Then define a program $P_{(V \backslash E)}$ as the parallel composition of emissions $\overline{s}v$ such that $v \in (V \backslash E)(s)$. Intuitively, this is the emission on an appropriate signal of all the values which are in V but not in E. We also let P_V stand for $P_{(V \backslash \emptyset)}$ where $\emptyset(s) = \emptyset$ for every signal s.

Definition 7 *With each action act, we associate a* minimal *program P_{act} that allows the action to take place:*

$$P_{act} = \begin{cases} 0 & \text{if } act = \tau \text{ or } act = N \\ \overline{s}v & \text{if } act = sv \text{ or } act = s?v \\ s(x).0, 0 & \text{if } act = \overline{s}v \\ P_{V \backslash E} & \text{if } act = (E, V) \end{cases}$$

Definition 8 (compatibility context and action) *A context Γ is compatible with an action act, written $(\Gamma, act) \downarrow$, if $\exists \Gamma'\ (\Gamma \oplus \Gamma') \downarrow$ and $\Gamma' \vdash P_{act}$.*

We can now introduce the concept of *typed* transition which is a transition labelled with an action *act* of a program typable in a context Γ such that Γ and *act* are compatible.

Definition 9 (typed transition) *We write $P \xrightarrow[\Gamma]{act} Q$ ($P \overset{act}{\underset{\Gamma}{\Rightarrow}} Q$) if: (1) $\Gamma \vdash P$, (2) $(\Gamma, act) \downarrow$, and (3) $P \xrightarrow{act} Q$ ($P \overset{act}{\Rightarrow} Q$, respectively).*

Next, we introduce the notion of *residual context* which is intuitively the context left after a typed transition. (the definition for the auxiliary actions is available in appendix B.5). First, we notice that given a (uniform) type σ and a value v we can define the minimum context $\Delta(v,\sigma)$ such that $\Delta(v,\sigma) \vdash v : \sigma$. Namely, we set $\Delta(s,\sigma) = s : \sigma$ and $\Delta(\mathsf{c}(v_1,\ldots,v_n)) = \Delta(v_1,\sigma_1) \oplus \cdots \oplus \Delta(v_n,\sigma_n)$ if $c : (\sigma_1,\ldots,\sigma_n) \to \sigma$. Notice that $\Delta(v,\sigma)$ is the empty context if $fn(v) = \emptyset$ and it is a neutral context if σ is non-affine.

Definition 10 (residual context) *Given a context Γ and a compatible and relevant action α, the residual context $\Gamma(\alpha)$ is defined as follows:*

$$\Gamma(\alpha) = \begin{cases} \Gamma & \text{if } \alpha = \tau \\ \uparrow \Gamma & \text{if } \alpha = N \\ (\Gamma, \mathbf{t} : \sigma') \ominus \Delta(v : \sigma') \oplus \{s : Sig_{u_5}(\sigma')\} & \text{if } \Gamma(s) = Sig_u(\sigma'), \alpha = \nu \mathbf{t} : \sigma' \overline{s} v, (1) \\ \Gamma \oplus \Delta(v,\sigma') \oplus \{s : Sig_{u_{out}}(\sigma')\} & \text{if } \Gamma(s) = Sig_u(\sigma'), \alpha = sv, (2) \end{cases}$$

*(1) $u_5 = \big(0,1,0\big) \cdot \big(0,0,0\big)^{\omega}$ if $u \in U(5)$ and it is neutral otherwise (*i.e.*, $u \in U(2)$). (2) u_{out} is the least usage of the same kind as u which allows to perform an output within the instant (always defined).*

The notion of residual context is instrumental to a precise statement of the way transitions affect the typing. First we notice that the type of expressions is preserved by the evaluation relation.

Lemma 11 (expression evaluation) *If $\Gamma \vdash e : \rho$ and $e \Downarrow v$ then $\Gamma \vdash v : \rho$.*

The following lemma records the effect of the substitution at the end of the instant.

Lemma 12 (substitution, end of instant) (1) *If $\Gamma \vdash [A(\mathbf{r})]$, $\Gamma' \vdash P_V$, and $(\Gamma \oplus \Gamma') \downarrow$ then $\uparrow (\Gamma \oplus \Gamma') \vdash V(A(\mathbf{r}))$.*
(2) *If moreover there are V', E such that $V, V' \Vdash\!\!- E$ then $V(A(\mathbf{r})) \approx V'(A(\mathbf{r}))$.*

Finally, the subject reduction theorem states that the residual of a typed transition is typable in the residual context (again, the residual context on auxiliary actions is defined in appendix B.5).

Theorem 13 (subject reduction) *If $P \xrightarrow[\Gamma]{act} Q$ then $\Gamma(act) \vdash Q$.*

Next we introduce a notion of *typed bisimulation* which refines the one given in definition 1 by focusing on typed processes and typed transitions. Let *Cxt* be the set of contexts and if $\Gamma \in Cxt$ let $Pr(\Gamma)$ be the set of programs typable in the context Γ.

Definition 14 (typed bisimulation) *A typed bisimulation is a function $\mathcal{R}$ indexed on Cxt such that for every context Γ, $\mathcal{R}_\Gamma$ is a symmetric relation on $Pr(\Gamma)$ such that: $P\, \mathcal{R}_\Gamma\, Q$, $P \xrightarrow[\Gamma]{\alpha} P'$, $bn(\alpha) \cap fn(Q) = \emptyset$ implies $\exists\, Q'$ ($Q \overset{\hat{\alpha}}{\underset{\Gamma}{\Rightarrow}} Q'$, $P'\, \mathcal{R}_{\Gamma(\alpha)}\, Q'$). We denote with $\approx^t$ the largest typed labelled bisimulation.*

An expected property of typed bisimulation is that it is a weaker property than untyped bisimulation: if we cannot distinguish two processes by doing arbitrary actions we cannot distinguish them when doing actions which are compatible with the typing.

Proposition 15 *If $P, Q \in Pr(\Gamma)$ and $P \approx Q$ then $P \approx^t_\Gamma Q$.*

We write $P \overset{\tau}{\underset{\Gamma}{\rightsquigarrow}} Q$ if $P \xrightarrow[\Gamma]{\tau} Q$ or $P = Q$. The following lemma states a strong commutation property of typed τ actions and it entails that typed bisimulation is invariant under τ-actions.

Lemma 16 (1) *If $P \xrightarrow[\Gamma]{\tau} P_i$ for $i = 1, 2$ then there is a Q such $P_i \overset{\tau}{\underset{\Gamma}{\rightsquigarrow}} Q$ for $i = 1, 2$.*

(2) *If $P \overset{\tau}{\underset{\Gamma}{\Rightarrow}} Q$ then $P \approx^t_\Gamma Q$.*

The second key property is that the computation at the end of the instant is deterministic and combining the two lemmas, we derive that typable programs are deterministic.

Lemma 17 *If $P \xrightarrow[\Gamma]{N} P_i$ for $i = 1, 2$ then $P_1 \approx^t_{\uparrow(\Gamma)} P_2$.*

Theorem 18 (determinacy) *If $P \overset{N}{\underset{\Gamma}{\Rightarrow}} \cdot \overset{N}{\underset{\Gamma'}{\Rightarrow}} \cdots \overset{N}{\underset{\Gamma'}{\Rightarrow}} P_i, i = 1, 2, \Gamma' = \uparrow \Gamma$ then $P_1 \approx^t_{\Gamma'} P_2$.*

5 Conclusion

The main contribution of this work is the identification of 5 kinds of usages in signal-based communication and of the rules that allow their *composition* while preserving determinacy. This goes well-beyond previous analyses for ESTEREL-like languages we are aware of that are essentially 'first-order' in the sense that signals are not treated as first-class values. Technically, we have shown that a typable process P is *deterministic*. This result builds on previous work by the authors [2, 4] on a mathematical framework to reason about the equivalence of programs which is comparable to the one available for the π-calculus.

References

[1] R. Amadio. The SL synchronous language, revisited. *Journal of Logic and Algebraic Programming*, 70:121-150, 2007.

[2] R. Amadio. A synchronous π-calculus. *Information and Computation*, 205(9):1470–1490, 2007.

[3] R. Amadio, I. Castellani and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Theoretical Computer Science*, 195:291-324, 1998.

[4] R. Amadio, M. Dogguy. Determinacy in a synchronous π-calculus. Technical Report, Université Paris 7, Laboratoire PPS, July 2007. To appear in *From semantics to computer science: essays in honor of Gilles Kahn*, Y. Bertot *et al* (eds.), CUP.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.

[6] F. Boussinot and R. De Simone. The SL synchronous language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.

[7] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*,50(1):1-102, 1987.

[8] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437-486, 1995.

[9] N. Kobayashi. Type systems for concurrent programs. In Proc. *10th Anniversary Colloquium of UNU/IIST*, Springer LNCS 2757, 2003.

[10] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems (TOPLAS), 21(5), 1999.

[11] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Proc. ACM Principles and Practice of Declarative Programming*, pages 82–93, 2005.

[12] R. Milner. Communication and concurrency. Prentice-Hall, 1989.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[14] Ph. Wadler. A Taste of Linear Logic. In Proc. *Mathematical Foundations of Computer Science*, SLNCS 711, pages 185-210, 1993.

A Typing examples

We consider two examples that are part of the folklore on synchronous programming (see, *e.g.*, [11]) and a third one that suggests that a certain form of single-assignment reference can be modelled in our framework.

Example 19 (cell) *We describe the behaviour of a generic cell that might be used in the simulation of a dynamic system. Each cell relies on three parameters: its state q, its own activation signal s, and the list ℓ of activation signals of its neighbours. The cell performs the following operations in a cyclic fashion: (i) it emits its current state along the activation signals of its neighbours, (ii) it waits till the end of the current instant (*pause*), and (iii) it collects the values emitted by its neighbours and computes its new state.*

$$\begin{array}{lcll} \mathit{Cell}(q,s,\ell) & = & \mathit{Send}(q,s,\ell,\ell) & \\ \mathit{Send}(q,s,\ell,\ell') & = & [\ell' \trianglerighteq \mathsf{cons}(s',\ell'')] & (\overline{s'}q \mid \mathit{Send}(q,s,\ell,\ell'')), \\ & & & \mathsf{pause}.\mathit{Cell}(\mathit{next}(q,!s),s,\ell) \end{array}$$

where next is a function that computes the following state of the cell according to its current state and the state of its neighbours. Assuming that the function next is invariant under permutations of the list of states, we would like to show that the evolution of the simulation is deterministic. To express this invariance, a natural idea is to treat the 'list' of distinct states as a 'set', i.e.*, as a list* quotiented *by a relation that identifies a list with any of its permutations.*

We now turn to the typing. Assume an inductive (non-affine) type State to represent the state of a cell and let $\sigma = \mathit{Sig}_u(\mathit{State})$ where $u = (\infty, 0, \infty)^\omega$ and $\sigma' = \mathit{List}_\infty(\sigma)$. Then we can require: $\mathit{Cell} : (\mathit{State}, \sigma, \sigma')$ and $\mathit{Send} : (\mathit{State}, \sigma, \sigma', \sigma')$. Because, the usage of the signals under consideration is $(\infty, 0, \infty)^\omega$, the type of their dereferenciation is $\mathit{Set}_\infty(\mathit{State})$ and therefore we must require $\mathit{next} : (\mathit{State}, \mathit{Set}_\infty(\mathit{State})) \to \mathit{State}$, which means that the result of the function next must be invariant under permutations of the list of (distinct) states.

Example 20 (synchronous data flow) *We provide an example of synchronous data-flow computation. The network is described by the program*

$$\nu s_2, s_3, s_4, s_5(\ A(s_1,s_2,s_3,s_4) \mid B(s_2,s_3,s_5,s_6) \mid C(s_4,s_5)\ )$$

$$\textit{where:} \begin{cases} A(s_1,s_2,s_3,s_4) & = s_1(x).(\overline{s_2}f(x) \mid s_3(y).(\overline{s_4}g(y) \mid \mathsf{pause}.A(s_1,s_2,s_3,s_4)),0),0 \\ B(s_2,s_3,s_5,s_6) & = s_2(x).(\overline{s_3}i(x) \mid s_5(y).(\overline{s_6}l(y)) \mid \mathsf{pause}.B(s_2,s_3,s_5,s_6)),0),0 \\ C(s_4,s_5) & = s_4(x).(\overline{s_5}h(x) \mid \mathsf{pause}.C(s_4,s_5)),0 \end{cases}$$

Assuming that at each instant at most one value is emitted on the input signal s_1, we would like to show that at each instant at most one value will be emitted on every other signal. This example suggests that we should introduce a notion of affine usage *in signals.*

We now turn to the typing. We assume an inductive type D of data *and let $\sigma = Sig_u(D)$, $\sigma_I = Sig_{u_I}(D)$, and $\sigma_O = Sig_{u_O}(D)$, where: $u = (1,1,0)^\omega$, $u_I = (0,1,0)^\omega$, and $u_O = (1,0,0)^\omega$. Then we can require: $A : (\sigma_I, \sigma_O, \sigma_I, \sigma_O)$, $B : (\sigma_I, \sigma_O, \sigma_I, \sigma_O)$, and $C : (\sigma_I, \sigma_O)$. The restricted signals $s_2, \ldots, s_5$ take the type σ and the overall system is well-typed with respect to the context $s_1 : \sigma_I, s_6 : \sigma_O$.*

Remark 21 (affinity vs. linearity) *With reference to the data flow example 20, one may notice that the type system guarantees determinacy by making sure that at every instant* at most one *value is emitted on every signal. One could consider a more refined type system that guarantees that* exactly one *value is emitted on a signal at every instant.*[4] *However, to obtain this system it is not enough to require that all linear hypotheses in the context are used in the typing. For instance, consider: $\nu s, s' : \sigma(A(s,s') \mid A(s',s))$ where: $\sigma = Sig_{(1,1,0)^\omega}$, $A : (\sigma, \sigma)$, and $A(s,s') = s().(\overline{s'} \mid \mathsf{pause}.A(s,s')), A(s,s')$. This program could be* linearly *typed but it is stuck at every instant. Following previous work (see,* e.g., *[9]), one way to address this problem is to partition signals in a finite set of regions and to order them. Then one designs typing rules that require that a reception on a signal belonging to a given region only guards (prefixes) emissions on signals belonging to higher regions.*

Example 22 (single-assignment references) *We introduce a kind of* single-assignment *references that allow for a shared memory among different threads while preserving determinacy. For simplicity, we look at references on some basic inductive type κ. The three basic operations are: (1) $\mathsf{newref}(s,e)\ P$ creates a reference s whose scope is P and assigns it the value resulting from the evaluation of e; (2) $\mathsf{read}(s,x).P$ reads the value v contained in the reference s and runs $[v/x]P$; and (3) $\mathsf{write}(s,e).P$ evaluates e and writes its value in the reference s. The written value will be available in the following instant. Reading and writing are non-blocking operations, moreover a value written at a given instant persists unless a following write operation occurs. To ensure determinacy, we have to guarantee that at any instant at most one value is written in a reference.*

We model this situation by associating with each reference s a pair of signals (s, s'). The first signal s has a usage of kind 2 (one write and arbitrarily many reads) while the signal s' has a usage of kind 5 (one write and one read during the instant). A reference s containing the value x is simulated by the following recursive program:

$$Ref(s,s',x) \quad = \overline{s}x \mid s'(y).\mathsf{pause}.Ref(s,s',y), Ref(s,s',x)$$

where the type of Ref is $(Sig_u(\kappa), Sig_{u'}(\kappa), \kappa)$ with $u = (1,\infty,\infty)^\omega$ and $u' = (0,1,0)^\omega$. Thus on the signal s, Ref emits the current value of the reference while on the signal s' it waits for the value for the next instant. The usages we assign to the signals s and s' guarantee that arbitrarily many threads can read the reference but at most one can write it at any given instant. Formally, we can translate the three basic operations on references described above as follows:

$$\begin{array}{ll} \langle \mathsf{newref}(s,e)\ P \rangle & = \nu s, s'\ (Ref(s,s',e) \mid \langle P \rangle), \\ \langle \mathsf{read}(s,x).P \rangle & = s(x).\langle P \rangle, 0, \\ \langle \mathsf{write}(s,e).P \rangle & = \overline{s'}e \mid \langle P \rangle\ . \end{array}$$

[4] In this system the 'else' branch of the input operator would become useless

Example 23 (clocks) *We consider a kind of clock that still allows for a deterministic execution.*[5] *The value of a clock is a natural number which is emitted on a signal, hence within an instant all threads can read the same clock value. At each instant, one or more threads may reset the clock value. The effect of this reset is visible in the following instant. To program a clock, we declare the unit type and the type of natural numbers:*

$$\begin{array}{ll} Unit_\infty() & = * \\ Nat_\infty() & = Z \parallel S\ of\ Nat() \end{array}$$

With each clock we associate a thread Clock whose behaviour and type is defined as follows:

$$\begin{array}{ll} Clock(s,r,n) & = \overline{s}n \mid \mathsf{pause}.Clock'(s,r,!r,n) \\ Clock & : (Sig_u(Nat), Sig_{u'}(Unit), Nat), \quad u = (1,\infty,\infty)^\omega, \quad u' = (\infty,0,1)^\omega \\ Clock'(s,r,\ell,n) & = [\ell \trianglerighteq \mathsf{nil}]Clock(s,r,S(n)), Clock(s,r,Z) \\ Clock' & : (Sig_u(Nat), Sig_{u'}(Unit), Set_1(Unit), Nat) \end{array}$$

Note that the typing guarantees that the thread Clock is the only one that can emit the clock signal s and read the reset signal r. On the other hand, another thread using the clock may read *the clock value on the signal s and may reset it in the following instant by emitting on the reset signal r.*

B Proofs

B.1 Proof of lemma 5

By induction on the typing rules. One uses several times the fact that $\oplus$ is associative and commutative both on types and contexts and the fact that the rules are formulated so that the conclusion still holds when the usages in the context Γ are increased (see, *e.g.*, the rule (var)).

B.2 Proof of lemma 6

The following lemma collects some preliminary remarks.

Lemma 24 (1) *If $\Gamma \vdash U : T$, $\Gamma' \vdash v : \rho$, $(\Gamma \oplus \Gamma') \downarrow$, and $x \notin dom(\Gamma)$ then* $(\Gamma \oplus \Gamma') \vdash [v/x]U : T$.

(2) *If $\Gamma \vdash v : \kappa$ then there is a neutral context Γ' such that $\Gamma' \vdash v : \kappa$ and $\Gamma = \Gamma' \oplus \Gamma''$.*

(3) *If $\Gamma \vdash v : \rho$ and $\rho = \rho_1 \oplus \cdots \oplus \rho_n$ then there exist $\Gamma_1, \ldots, \Gamma_n$ such that $\Gamma_1 \oplus \cdots \oplus \Gamma_n = \Gamma$ and $\Gamma_i \vdash v : \rho_i$ for $i = 1, \ldots, n$.*

Proof (1) If $x \in FV(U)$ then the only possibility is that $x \in FV(e)$ where $\underline{\overline{s}e}$ is a sub-term of U. But then one can type $\underline{\overline{s}[v/x]e}$ exactly as one types $\underline{\overline{s}e}$. So $\Gamma \vdash [v/x]U : T$ and we conclude by weakening.

(2) We proceed by induction on v. For the inductive step, we use the fact that if $\mathsf{c}(v_1, \ldots, v_n)$ has a neutral type then the v_i must have a neutral type too.

(3) If the type ρ is neutral then $\rho = \rho_1 = \cdots = \rho_n$. By (2), we can find a neutral context Γ' such $\Gamma' \vdash v : \rho$ and $\Gamma' \oplus \Gamma'' = \Gamma$. Then it suffices to take $\Gamma_1 = \Gamma' \oplus \Gamma''$ and $\Gamma_i = \Gamma'$

[5] Note that in the usual semantics of timed automata, the fact that two processes may atomically *read and reset* the same clock may produce race conditions.

for $i = 2, \ldots, n$. If the type ρ is affine and either an inductive type or a set type then we must have $n = 1$ and the assertion follows immediately. Finally, if the type ρ is affine and a signal type then the usages of the signal in the types $\rho_1, \ldots, \rho_n$ allow to construct directly the contexts $\Gamma_1, \ldots, \Gamma_n$. □

Next, to prove the substitution lemma we proceed by induction on the typing of U.

(var) Suppose $\Gamma, y : Op_u(\sigma) \vdash y : Op_{u'}(\sigma)$ with $u \geq u'$.

• If $\Gamma = \Gamma'', x : \rho$ and $x \neq y$ then $((\Gamma'', y : Op_u(\sigma)) \oplus \Gamma')(y) = Op_{u''}(\sigma)$ with $u'' \geq u$. Hence, by (var), $(\Gamma'', y : Op_u) \oplus \Gamma' \vdash y : Op_{u'}$.

• If $x = y$ then $[v/x]y = v$. If Op is not Sig then $u = u'$. By hypothesis, $\Gamma' \vdash v : Op_u(\sigma)$ and by weakening $\Gamma'' \oplus \Gamma' \vdash v : Op_u(\sigma)$. On the other hand, if Op is Sig then, by (var), $(\Gamma'' \oplus \Gamma') \vdash v : Op_u(\sigma)$.

(k) If k is a constant then apply weakening. Otherwise, suppose $\Gamma, x : \rho = \Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n$ with $\Gamma_i \vdash e_i : \sigma_i$, $i = 1, \ldots, n$. Let $I = \{i \in \{1, \ldots, n\} \mid x \in dom(\Gamma_i)\}$. If $i \in I$ then assume $\Gamma_i = \Gamma_i'', x : \rho_i$. We have $\rho = \oplus_{i \in I} \rho_i$. By lemma 24(3), we can find Γ_i' such that $\Gamma_i' \vdash v : \rho_i$ for $i \in I$ and $\Gamma' = \oplus_{i \in I} \Gamma_i'$. If $i \notin I$ then $\Gamma_i \vdash [v/x]e_i : \sigma_i$, (cf. lemma 24(1)), and if $i \in I$ then $(\Gamma_i \oplus \Gamma_i') \vdash [v/x]e_i : \sigma_i$, by inductive hypothesis.

This kind of argument is repeated several times for the remaining rules. As already pointed out in the proof of the weakening lemma 5, another important point is that the rules are built so that adding extra capabilities to the hypotheses in the context does not affect the conclusion. We just look in some detail at the rule $[var_{sig}]$ in the case where $\Gamma, s : Sig_{xy^\omega}(\sigma) \vdash [s] : Sig_u(\sigma)$, $y^\omega \geq u$, $\Gamma' \vdash s' : Sig_{xy^\omega}(\sigma)$ and $(\Gamma \oplus \Gamma') \downarrow$. Then $\Gamma'(s) = s' : Sig_{u'}(\sigma)$ with $u' \geq xy^\omega$. Hence $\uparrow(u') \geq y^\omega \geq u$. □

B.3 Proof of lemma 11

By induction on the evaluation $e \Downarrow v$. If e is a signal s or a constant c then $e = v$ and the conclusion is immediate. So suppose: $e = k(e_1, \ldots, e_n)$, $k : (\sigma_1, \ldots, \sigma_n) \to \sigma$, $\Gamma = \Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n$, $\Gamma_i \vdash e_i : \sigma_i$, and $e_i \Downarrow v_i$, for $i = 1, \ldots, n$. By inductive hypothesis, $\Gamma_i \vdash v_i : \sigma_i$, for $i = 1, \ldots, n$. If k is a constructor c then $v = \mathsf{c}(v_1, \ldots, v_n)$ and $\Gamma \vdash v : \sigma$ by the rule (k). If k is a function f then again by the rule (k), $\Gamma \vdash f(v_1, \ldots, v_n) : \sigma$ and, by hypothesis on f, we have that $f(v_1, \ldots, v_n) \Downarrow v$ and $\Gamma \vdash v : \sigma$. □

B.4 Proof of lemma 12

(1) The effect of $V(A(\mathbf{r}))$ is to replace each of occurrence of $!s$ in $\mathbf{r}$ with $V(s)$. First notice that if $!s$ occurs in $\mathbf{r}$ then its usage cannot be of kind 5. Moreover, if it is of kind 1 or 2 then we can have several occurrences of $!s$ in $\mathbf{r}$ and the type of the values emitted on the signal must be non-affine. Notice that to type a non-affine value, we just need a non-affine context and since non-affine types are (exactly the) neutral types, we can use this context as many times as needed. On the other hand, if the signal is of kind 3 or 4 then the values emitted on the signal can be affine but there can be no more than one occurrence of $!s$ in $\mathbf{r}$.

Following these preliminary considerations, we proceed by case analysis on the rules $[!_{Set}]$ and $[!_{List}]$. In each case, one has a judgement of the shape:

$$\Gamma, s : Sig_u(\sigma) \vdash [!s] : Op_x(\sigma)$$

knowing that $\Gamma' \vdash V(s) = [v_1; \ldots; v_n] : Op_x(\sigma)$,

(2) By definition, $V(A(r_1, \ldots, r_n)) = A(V(r_1), \ldots, V(r_n))$. Suppose $A : (\sigma_1, \ldots, \sigma_n)$. We know that $v_i \sim_{\sigma_i} u_i$ entails that $A(v_1, \ldots, v_n) \approx A(u_1, \ldots, u_n)$. Hence, it is enough to show that that $V(r_i) \sim_{\sigma_i} V'(r_i)$ for $i = 1, \ldots, n$. We proceed by induction on the structure of r. If r is a signal or a constant then by definition $r \sim_{\sigma_i} r$. If r is of the shape $!s$ then we analyse the kind of usage of s. If it is of kind 2 or 4 then $V(s) = V'(s)$ (there is at most one value in the lists). If it is of kind 1 or 3 then $V(s)$ and $V'(s)$ are equal up to permutation, and we rely on the definition of $\sim$ on set types. Finally, if $r = k(\mathbf{r})$ we apply the inductive hypothesis plus the definition of $\sim$ on constructors if k is a constructor and the hypothesis on the functions if k is a function.

B.5 Residual context on auxiliary actions

We specify the notion of *residual context* on auxiliary actions. The definition for the actions $s?v$ is similar to the one for the actions sv. On the other hand, for the actions (E, V), we have to analyse how a program exports and imports usages at the end of the instant. For instance, consider $P = \overline{s_1}t_1 \mid \overline{s_2}t_2 \mid A(!s_1)$, and suppose $P \xrightarrow[\Gamma]{(E,V)} A(V(s_1))$ where:

$$E \;=\; [\{t_1\}/s_1, \{t_2\}/s_2] \qquad V \;=\; [[t_1; t_3]/s_1, [t_4; t_2]/s_2] \;.$$

The function E represents what P emits, the function V represents what P assumes to be emitted, moreover looking at the context Γ, we may determine what the process P may receive at the end of the instant (note that P may receive what it emits and that a value with an affine typing can be received at most once). In computing the residual context, we have to subtract what is exported to the environment while adding what is imported from it. Going back to our example, clearly the context Γ must specify that P may receive on s_1 at the end of the instant. Suppose moreover that it specifies that P may not receive on s_2. Then in computing the residual context, we have to subtract the usage for t_2 which is exported to the environment while adding the usage for t_3 which is received from it. Following these considerations, we define:

$$\Delta(E, \Gamma) \;=\; \oplus\{\Delta(v, \lambda) \mid \Gamma(s) = Sig_u(\lambda), v \in E(s), u(0)_3 \neq 1\} \quad \text{(export)}$$

$$\Delta(V, \Gamma) \;=\; \oplus\{\Delta(v, \sigma) \mid \Gamma(s) = Sig_u(\sigma), v \in V(s), u(0)_3 \neq 0\} \quad \text{(import)}$$

Note that in the 'exported context' $\Delta(E, \Gamma)$ we only care about usages of values of affine type, as otherwise $\Delta(v, \kappa)$ is neutral. On the other hand, in the 'imported context' we look at all the values regardless of their type. Indeed, v might have a neutral type but contain a fresh signal name and then we need to import a neutral context to type it. Also note that in the following definition 25, we actually focus only on the values that are *not* emitted (in E).

Definition 25 (residual context on auxiliary actions) *Given a context Γ and an auxiliary action aux the residual context $\Gamma(aux)$ is defined as follows where u_5 is as in definition 10:*

$$\Gamma(aux) = \begin{cases} (\Gamma \ominus \{s : Sig_{u_5}(\sigma')\}) \oplus \Delta(v, \sigma') & \text{if } \Gamma(s) = Sig_u(\sigma'), aux = s?v, \text{ and (1)} \\ (\uparrow \Gamma \ominus \Delta(E, \Gamma)) \oplus \Delta(V', \Gamma) & \text{if } aux = (E, V) \text{ and } V \backslash E = V' \end{cases}$$

B.6 Proof of theorem 13

We proceed by induction on the proof of the transition and by case analysis on the action *act* which is performed.

(sv) There is just 1 rule to consider: (in). Suppose $\Gamma(s) = Sig_u(\sigma')$. The definition of the residual context provides an additional context $\Delta(v, \sigma') \oplus \{s : Sig_{u_{out}}(\sigma')\}$ which is just what is needed to type $\overline{s}v$.

$(s?v)$ There are 3 rules to consider: (in_{aux}), $(comp)$, and (ν). We just look at the first one. Suppose $(\Gamma_1 \oplus \Gamma_2) \vdash s(x).P, K$, $\Gamma_1 \vdash s : Sig_u(\sigma')$, $u(0)_2 \neq 0$, $\Gamma_2, x : \sigma' \vdash P$, and $\Gamma_1 \oplus \Gamma_2 \vdash [K]$. Note that necessarily $u \geq u_{in}$. By construction, $\Delta(v, \sigma') \vdash v : \sigma'$. By the substitution lemma 6, $\Gamma_2 \oplus \Delta(v, \sigma') \vdash [v/x]P$ and then it is enough to apply weakening to get the residual context.

$(\nu \mathbf{t} : \sigma\ \overline{s}v)$ There are 5 rules to consider: (out), with a special treatment for kind 5, $(\underline{out})$, (ν_{ex}), $(comp)$, and (ν).

(τ) There are 8 rules to consider: $(synch)$, (rec), $(=_i^{sig})$, $(=_i^{ind})$, $(comp)$, and (ν) for $i = 1, 2$ We just look at the first two.

$(synch)$ Suppose: $P_1 \xrightarrow{\nu \mathbf{t}:\rho \overline{s}v} P_1'$, $P_2 \xrightarrow{s?v} P_2'$, $\Gamma_i \vdash P_i$, for $i = 1, 2$, and $(\Gamma_1 \oplus \Gamma_2)(s) = Sig_u(\sigma')$. By inductive hypothesis, we have:

$$(\Gamma_1, \mathbf{t} : \rho) \ominus \Delta(v, \sigma') \oplus \{s : Sig_{u_5}(\sigma')\} \vdash P_1' \quad \text{and}$$
$$(\Gamma_2 \oplus \Delta(v, \sigma') \ominus \{s : Sig_{u_5}(\sigma')\} \vdash P_2'$$

Recall that here u may be of kind 2 or 5 and that in the first case u_5 is neutral. In both cases, we get $(\Gamma_1 \oplus \Gamma_2), \mathbf{t} : \rho \vdash (P_1' \mid P_2')$, and we conclude applying the typing rule (ν).

(rec) Suppose $A : (\sigma_1, \ldots, \sigma_n)$, $\Gamma_i \vdash e_i : \sigma_i$, $e_i \Downarrow v_i$, for $i = 1, \ldots, n$. By lemma 11, $\Gamma_i \vdash v_i : \sigma_i$. By hypothesis, we know that if $A(x_1, \ldots, x_n) = P$ then $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P$. Thus, by iterating the substitution lemma 6, we get, as required, $\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash [v_1/x_1, \ldots, v_n/x_n]P$.

(E, V) There are 5 rules to consider: (0), $(reset)$, $(\underline{reset})$, $(cont)$, and (par). We focus on the last two.

$(cont)$ Suppose $s(x).P, K \xrightarrow{(\emptyset, V)} V(K)$ and $\Gamma \vdash s(x).P, K$. Then $\Gamma \vdash [K]$. We rely on lemma 12(1). We build the context Γ' in the lemma by taking $\Gamma' = \Delta(V, \Gamma)$ which is uniform added to a context Γ'' which just provides the usages to emit in the first instant the values in V on the signals in $dom(V)$.

(par) Suppose: $\Gamma = (\Gamma_1 \oplus \Gamma_2)$, $\Gamma \vdash (P_1 \mid P_2)$, $(P_1 \mid P_2) \xrightarrow{(E_1 \cup E_2), V} (P_1' \mid P_2')$, $\Gamma_i \vdash P_i$, $P_i \xrightarrow{(E_i, V)} P_i'$, for $i = 1, 2$. Following the definition of residual context, define for $i = 1, 2$:

$$\begin{array}{ll} Exp_i = \Delta(E_i, \Gamma_i) & Exp_{1,2} = \Delta(E_1 \cup E_2, \Gamma_1 \oplus \Gamma_2) \\ Imp_i = \Delta(V \backslash E_i, \Gamma_i) & Imp_{1,2} = \Delta(V \backslash (E_1 \cup E_2), \Gamma_1 \oplus \Gamma_2) \\ \Gamma_i' = \uparrow \Gamma_i \ominus Exp_i \oplus Imp_i & \Gamma' = \uparrow (\Gamma_1 \oplus \Gamma_2) \ominus Exp_{1,2} \oplus Imp_{1,2} \end{array}$$

We want to show $\Gamma' = \Gamma_1' \oplus \Gamma_2'$. We proceed, by analysing the contribution of each value $v \in V(s)$ such that $\Gamma(s) = Sig_u(\sigma)$ to the computation of Imp_i, $Imp_{1,2}$, Exp_i, and $Exp_{1,2}$. We use the notation, *e.g.*, $Imp_1(v)$ to denote the contribution of the value v to the computation of the context Imp_1.

• If σ is non-affine then, for $i = 1, 2$, Imp_i, and $Imp_{1,2}$ are neutral contexts while Exp_i and $Exp_{1,2}$ are empty contexts. Up to symmetries, v can be received either by (i) Γ_i, $i = 1, 2$ or (ii) Γ_1 and Γ_2 and emitted either by (i) $E_1 \cap E_2$, or (ii) $E_1 \backslash E_2$, or (iii) $E_2 \backslash E_1$, or by (iv) the environment. One proceeds by case analysis (8 situations).

• If σ is affine then the usage u must be of kind 3 or 4 and at the end of the instant the signal s may be read, exclusively, either by (i) Γ_i, $i = 1, 2$ or by (ii) the environment. On the other hand, v may be emitted either by (i) $(E_1 \cap E_2)$, or by (ii) $(E_1 \backslash E_2)$, or by (iii) $(E_2 \backslash E_1)$ or by (iv) $(V \backslash (E_1 \cup E_2))$. If $v \in (E_1 \cap E_2)(s)$ then $\Delta(v, \sigma)$ must be neutral for otherwise the addition is not defined. One then proceeds by case analysis (8 situations). Note that if the environment receives v then the import contexts $Imp_i, Imp_{1,2}$ are empty while if Γ_i receives v then Exp_i is empty.

(N) There is just 1 rule to consider: $(next)$. Suppose $\Gamma \vdash P$ and $P \succeq \nu \mathbf{s} : \rho\ P''$. Clearly, a typing of, say, $(\nu s : \rho\ Q_1) \mid Q_2$ can be transformed into a typing of $\nu s : \rho\ (Q_1 \mid Q_2)$. Thus $\Gamma \vdash \nu \mathbf{s} : \rho\ P''$ and $\Gamma, \mathbf{s} : \rho \vdash P''$. By definition of the rule $(next)$, $P'' \xrightarrow{(E,V)} P'$ with $V \Vdash E$. By inductive hypothesis and weakening, $\uparrow (\Gamma, \mathbf{s} : \rho) \vdash P'$. Thus $\uparrow (\Gamma) \vdash \nu \mathbf{s} : \uparrow \rho\ \vdash P'$. □

B.7 Proof of proposition 15

We show that the following indexed relation is a typed bisimulation:

$$P\ \mathcal{R}_\Gamma\ Q \quad \text{if} \quad P, Q \in Pr(\Gamma) \text{ and } P \approx Q\ .$$

Suppose $P\ \mathcal{R}_\Gamma\ Q$, $P \xrightarrow[\Gamma]{\alpha} Q$, and $bn(\alpha) \cap fn(Q) = \emptyset$. Then:

$P \xrightarrow{\alpha} P'$	(by definition of typed transition)
$\Gamma(\alpha) \vdash P'$	(by subject reduction)
$Q \stackrel{\alpha}{\Rightarrow} Q', P' \approx Q'$	(by untyped bisimulation)
$\Gamma(\alpha) \vdash Q'$	(by subject reduction)

Hence we can conclude that $P'\ \mathcal{R}_{\Gamma(\alpha)}\ Q'$. □

B.8 Proof of lemma 16

(1) An inspection of the labelled transition system in table 2 reveals that two τ reductions may superpose only if they are produced by two synchronisations on the same signal name, say s. In this case, s must have a usage of kind 2 or 5. In a usage of kind 2, the typing guarantees that there is at most one value emitted on s so that we are roughly in the following situation:

$$P = C[s(x).P_1, Q_1 \mid s(x).P_2, Q_2 \mid \overline{s}e]$$

Because a signal emission persists within an instant, it is possible to close the diagram in one step. On the other hand, in a usage of kind 5 there can be at most one receiver and therefore no superposition may arise.

(2) We show that $\overset{\tau}{\underset{\Gamma}{\rightsquigarrow}}$ is a typed bisimulation. If $P = Q$ nothing needs to be proved. So suppose $P \xrightarrow[\Gamma]{\tau} Q$. Clearly, P can weakly simulate all actions Q may perform just by performing initially an extra τ step. So suppose $P \xrightarrow[\Gamma]{\alpha} P'$. Note that $\alpha \neq N$ since P may perform a τ action.

$\alpha = \tau$ In this case, we apply (1) noticing that $\overset{\tau}{\underset{\Gamma}{\rightsquigarrow}} \subseteq \overset{\tau}{\underset{\Gamma}{\Rightarrow}}$.

$\alpha = sv$ In this case, $P' = (P \mid \overline{s}v)$ and we can close the diagram by performing $Q \xrightarrow{sv} (Q \mid \overline{s}v)$.

$\alpha = \nu \mathbf{t} \overline{s}v$ Again, because a value emitted on a signal persists, it is equivalent to use it in an internal synchronisation and then again to extrude the value to the environment or the other way around. □

B.9 Proof of lemma 17

By subject reduction we know that $\uparrow(\Gamma) \vdash P_i$. If we can show that $P_1 \approx P_2$ then by proposition 15 we can conclude. According to the rule (*next*) of the labelled transition system, we must have for $i = 1, 2$:

$$P \succeq \nu \mathbf{s}_i\ P', \quad \mathbf{s}_1 \text{ permutation of } \mathbf{s}_2, \qquad P' \xrightarrow{E, V_i} P_i'', \quad V_i \Vdash E, \qquad P_i = \nu \mathbf{s}_i P_i'' \ .$$

Then lemma 12(2) and fact 2 guarantee that $P_1'' \approx P_2''$ and $P_1 \approx P_2$. □

B.10 Proof of theorem 18

The proof is a direct diagram chasing relying on lemma 16(2), 17, and the definition of typed bisimulation. □